BAYES-LIN: An object-oriented environment for Bayes linear local computation

Darren J. Wilkinson*

February 2, 2008

The latest version of the BAYES-LIN software and documentation (including the latest version of this document), can be obtained from the BAYES-LIN WWW page:

`\protect\vrule width0pt\protect\href{http://www.ncl.ac.uk\st`

Abstract

BAYES-LIN is an extension of the LISP-STAT object-oriented statistical computing environment, which adds to LISP-STAT some object prototypes appropriate for carrying out local computation *via* message-passing between clique-tree nodes of Bayes linear belief networks. Currently the BAYES-LIN system represents a rather low-level set of tools for a back-end computational engine, together with diagnostic graphics for understanding the effects of adjustments on the moral graph. A GUI front end, allowing interactive formulation of DAG models could be easily added, but is currently missing from the system. This document provides a very brief introduction to the system, by means of a work-through of two example computations, followed by a list of variables, functions, objects and methods provided by the system.

*Department of Statistics, University of Newcastle, Newcastle upon Tyne, NE1 7RU, ENGLAND. Email: `d.j.wilkinson@ncl.ac.uk` WWW: `http://www.ncl.ac.uk/~ndjw1/`

Contents

1 Introduction

1.1 Bayes linear methods

Bayes linear methods are a form of Bayesian statistics, which acknowledge the difficulties associated with the full modelling, specification, and conditioning required by distributional Bayesian statistics, and instead try to make best possible use of partial specifications, based on means, variances and covariances. Unsurprisingly, much of the theory is formally identical to inference in multivariate Gaussian Bayesian networks, but interpretation of results is generally different. This document assumes a working knowledge of the basic tools of the Bayes linear methodology. An introduction to Bayes linear methods is given in [1]. An introduction to (non-local) computational issues can be found in [6]. The foundations of the theory are discussed in [5], [3], and [2]. On-line, an introduction to the theory can be found in [4], from the Bayes Linear Methods WWW home page: `http://fourier.dur.ac.uk:8000/stats/bayeslin/`

1.2 LISP-STAT

LISP-STAT is an interpreted, object-oriented environment for statistical computing, described in [8]. This document assumes a working knowledge of LISP-STAT, and the basics of object-oriented programming. On-line, LISP-STAT information is available from the LISP-STAT WWW home page: `http://www.stat.umn.edu/~luke/xls/xlsinfo/xlsinfo.html`

1.3 Local computation

BAYES-LIN carries out local computation *via* message-passing between adjacent nodes of a clique-tree representing the statistical model of interest. Again, local computation in Bayesian networks is a huge area, and this document assumes a working knowledge of graphical models, conditional independence and some of the ideas behind local computation. The best introduction to all of these areas is [7]. In particular, Chapter 3 of that volume deals with all of the relevant graph-theoretic concepts, and Section 7.2 gives an introduction to graphical Gaussian models.

1.4 Installing and running BAYES-LIN

You need a working LISP-STAT system installed before you attempt to install BAYES-LIN. The following instructions are for a UNIX system with an XLISP-STAT installation, but installing on other systems should be similar. Note that the graphics work best on systems with at least a 16 bit colour display. If you only have an 8 bit display (256 colours), make sure that most are free for use by BAYES-LIN. The graphics will not work on displays poorer than 8 bit colour. Create a new directory for the BAYES-LIN system. Download the BAYES-LIN software from the BAYES-LIN WWW page: `http://www.ncl.ac.uk/~ndjw1/bayeslin/` and put into the new directory. In this new directory type:

```
% gunzip blin01a.tar.gz
% tar -xvf blin01a.tar
% gzip blin01a.tar
```

You should then be able to run LISP-STAT with the BAYES-LIN extensions simply by running

```
% xlispstat
```

from within this directory. You can check that the extensions are loaded by typing in some of the following commands in the LISP-STAT listener window.

```
> (help 'create-tree-node)
> (send moral-node-proto :help)
> (send tree-node-proto :help :observe)
```

In general, to make sure the extensions are loaded, use the expression

```
> (require "bayeslin")
```

When you are satisfied that the extensions are loaded, exit BAYES-LIN.

```
> (exit)
```

In order to run the examples, simply call LISP-STAT with the example as first argument. *eg.*

```
% xlispstat ex-dlm
```

or

```
% xlispstat ex-mdlm
```

These two examples will be explained in the following sections.

2 A "toy" dynamic linear model

2.1 Description of the model

The BAYES-LIN code for this example can be found in the file `ex-dlm.lsp`, which is part of the standard BAYES-LIN distribution. The example concerns a very simple model for 3 observations in time. The model can be written in the form of a locally constant DLM.

$$X_t = \theta_t + \nu_t$$
$$\theta_t = \theta_{t-1} + \omega_t$$

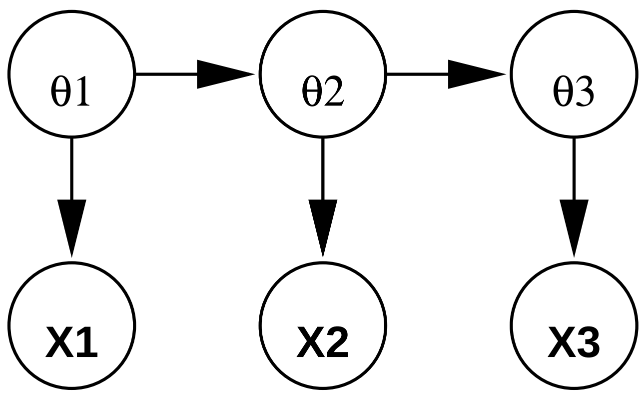
X_t denotes the observation at time t ($t = 1,2,3$), which is dependent on the *state* of the system at time t, θ_t. The variables ν_t and ω_t are incidental noise terms. The model is initialised by specifying beliefs about the initial state of the system; in this case, $\mathrm{E}(\theta_1) = 1$, $\mathrm{Var}(\theta_1) = 1$, and the variance of the noise terms; in this case, $\mathrm{Var}(\omega_t) = \mathrm{Var}(\nu_t) = 1$. Of course, for such a simple model, a non-local analysis is trivial, since the expectation vector for the entire system, and the variance matrix for the entire system can be written down and worked with directly. $\mathrm{E}(\theta_1,\theta_2,\theta_3,X_1,X_2,X_3) = (1,1,1,1,1,1)^{\mathrm{T}}$,

$$\mathrm{Var}(\theta_1,\theta_2,\theta_3,X_1,X_2,X_3) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 1 & 2 & 2 \\ 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 1 & 1 & 2 & 1 & 1 \\ 1 & 2 & 2 & 1 & 3 & 2 \\ 1 & 2 & 3 & 1 & 2 & 4 \end{pmatrix}$$

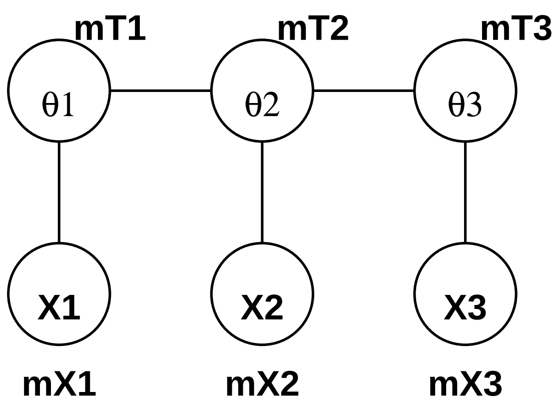
However, for the example in the next section, such explicit non-local analysis will not be possible.

2.2 Graphical models

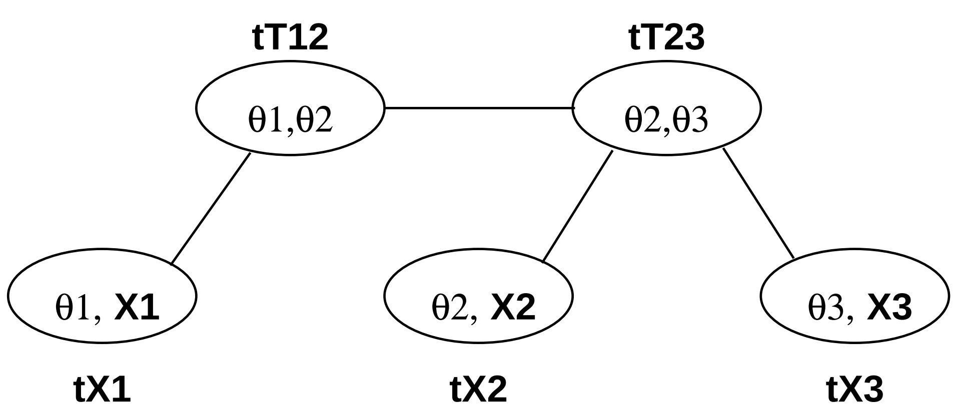
The graph for this model is shown below.



Since there are no unmarried parents, this graph can be moralised simply by dropping arrows.



Now, since there are no cycles, this graph is already triangulated, so the clique tree may be formed as follows.



2.3 Defining the clique-tree

BAYES-LIN carries out computation on the clique-tree, and displays results on the moral graph. Therefore, both need to be introduced to the BAYES-LIN system. Since all information and com-

putations are carried out on the clique-tree (in fact, computations can be carried out without defining a moral graph at all), this is defined first. Appropriate code for defining the tree nodes is shown below.

```
(create-tree-node 'tX1 '(t1 x1) #(1 1) #2a((1 1) (1 2)) '(tT12))
(create-tree-node 'tX2 '(t2 x2) #(1 1) #2a((2 2) (2 3)) '(tT23))
(create-tree-node 'tX3 '(t3 x3) #(1 1) #2a((3 3) (3 4)) '(tT23))
(create-tree-node 'tT12 '(t1 t2) #(1 1) #2a((1 1) (1 2)) '(tX1 tT23))
(create-tree-node 'tT23 '(t2 t3) #(1 1) #2a((2 2) (2 3)) '(tT12 tX2 tX3))
```

The global function `create-tree-node` is used to define each node in turn. The function expects five arguments. The first argument is a symbol to point to the resulting tree-node object. The second is a list of variables which the node contains. The third and fourth are the expectation vector and variance matrix for the variable list, and the fifth is a list of neighbouring tree nodes. Next, the moral nodes are defined.

2.4 Defining the moral graph

```
(create-moral-node 'mX1 '(x1) 'tX1 "mX1" '(mT1))
(create-moral-node 'mX2 '(x2) 'tX2 "mX2" '(mT2))
(create-moral-node 'mX3 '(x3) 'tX3 "mX3" '(mT3))
(create-moral-node 'mT1 '(t1) 'tX1 "mT1" '(mX1 mT2))
(create-moral-node 'mT2 '(t2) 'tX2 "mT2" '(mX2 mT1 mT3))
(create-moral-node 'mT3 '(t3) 'tX3 "mT3" '(mX3 mT2))
```

The global function `create-moral-node` is used to define each node in turn. The first is a symbol to bind the object to. The second is a variable list. The third is a clique-tree node which contains all of the variables at this node (such a node always exists). The fourth is a string to be used for plotting purposes, and the fifth is a list of neighbouring moral graph nodes. Next, some plotting positions are defined by sending a `:location` message to each moral node object.

```
(send mX1 :location '(0.2 0.8))
(send mX2 :location '(0.5 0.8))
(send mX3 :location '(0.8 0.8))
(send mT1 :location '(0.2 0.2))
(send mT2 :location '(0.5 0.2))
(send mT3 :location '(0.8 0.2))
```

This step may be omitted if one is not interested in plotting of results. The locations are on a $(0,1)$ scale for x and y coordinates, respectively. The origin is the top-left of the plot window. The model is now completely specified. Before carrying out adjustment, we create plot windows to show diagnostic information.

2.5 Adjustments

```
(create-moral-plot 'myplot)
(create-global-moral-plot 'myplot2)
```

This creates a plot window with the name `myplot` to show partial adjustment information, and another, `myplot2`, to show global adjustment information. Note that high-quality colour Encapsulated PostScript output is produced for each plot after each redraw of the screen, and stored in the files `mpw.eps` and `gmpw.eps` respectively.

We are now in a position to carry out adjustments. Suppose that variable X_1 is observed to be $x_1 = 3$. This information can be introduced into the graph by sending the following message to the appropriate moral graph node.

```
(send mx1 :observe '(x1) #(3))
```

In general, one can observe a list of variables, provided all variables are contained in the moral graph node receiving the message. The message is passed on to the appropriate clique-tree node, and then propagated around the clique-tree. We can tell our plot object to gather information from the tree for display, as follows.

```
(send myplot :record)
```

The plot should now show how information flows around the moral graph (more on this later). Note that although information has been introduced into the graph, it has not been *absorbed* into it, and that further information can not be introduced until it has. This can be understood by sending some messages to the graph, and looking at the return values. If the expectation and variance of the first observable node is examined

```
(send mx1 :ex)
(send mx1 :var)
```

it can be seen that it retains its *a priori* values. However, one can also ask for *adjusted* expectations and variances.

```
(send mx1 :aex)
(send mx1 :avar)
```

Similar queries can be sent to the third moral graph node.

```
(send mx3 :ex)
(send mx3 :var)
(send mx3 :aex)
(send mx3 :avar)
```

When we are finished examining the effects of the current adjustment, and wish to add further information into the graph, the current information should be absorbed.

```
(send mx1 :absorb)
```

The absorbing makes the adjusted information the new prior information, ready for the next adjustment. This can be verified by looking at the new expectation and variance for the third observation.

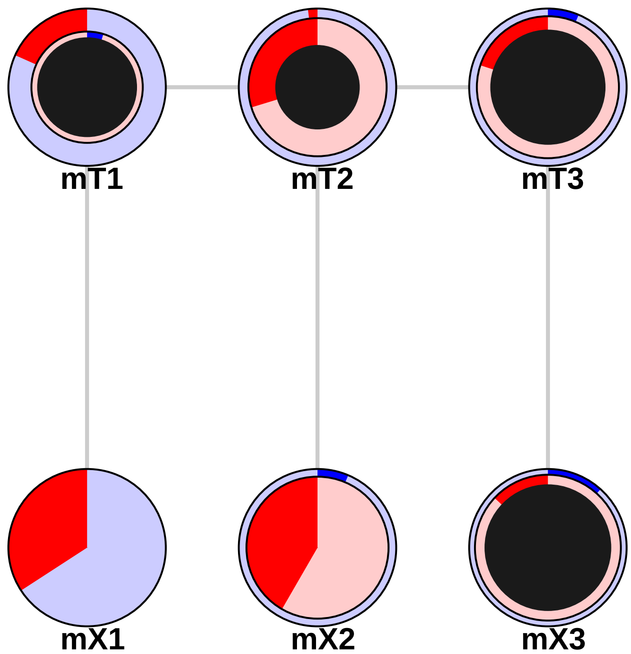
```
(send mx3 :ex)
(send mx3 :var)
```

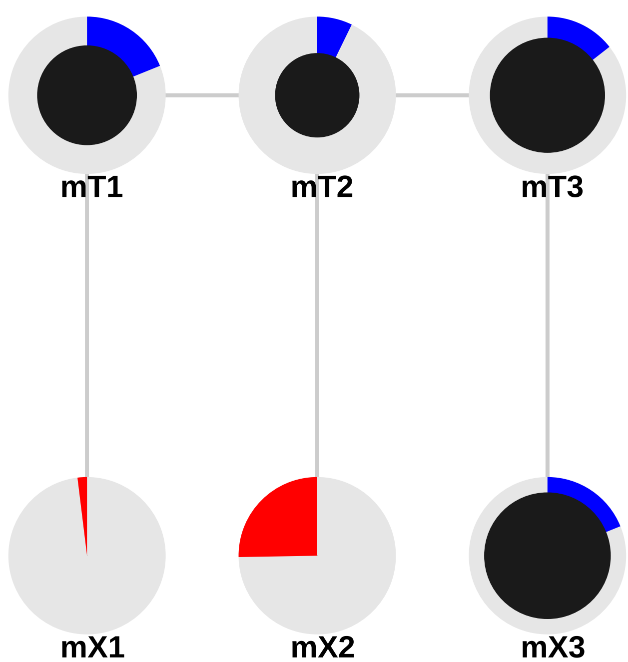
Finally, we can introduce new information, record it, and then absorb it, before examining the results.

```
(send mx2 :observe '(x2) #(-1))
(send myplot :record)
(send mx2 :absorb)

(send mx3 :var)
(send mx3 :ex)
```

The two plot windows should now look similar to the following.





2.6 Interpreting the graphics window

Whenever an observation is made and recorded, a portion is removed from the outside of each node. The area removed is proportional to the variance resolved by the adjustment. Consequently, the radius removed is proportional to the standard deviation resolved. Therefore, when a node is fully observed, there is no dark centre left remaining. For other nodes, the size of the dark centre is proportional to the proportion of original uncertainty left remaining. For multivariate nodes, the Bayes linear concept of *resolution* is used.

The additional red and blue shadings give an indication of the changes in expectation, relative to *a priori* uncertainty. Red shadings indicate changes larger than expected, and blue shadings represent changes in expectation smaller than expected*. The amount of red and blue shading increases as the "degree of surprisingness" increases. The amount of shading is a transformation of the Bayes linear concept of *size-ratio*. The transformation can be user-specified by redefining the plot-object's `:sr-map` method appropriately.

3 Computation for a large multivariate DLM

3.1 Description of the model

The BAYES-LIN code for this example can be found in the file `ex-mdlm.lsp`, which is part of the standard BAYES-LIN distribution. The following data represent weekly sales of six soft drinks packs from a wholesale depot.

```
 51 27 1  4  6 3
113 55 0  7 15 4
103 71 0 10 16 7
  .  . .  .  . .
  .  . .  .  . .
```

Clearly a multivariate time series model is required for such data. The following multivariate locally constant DLM is adopted.

$$X_t = \theta_t + \nu_t$$
$$\theta_t = \theta_{t-1} + \omega_t$$

This is the same model as used in the last example, but here all of the variables denote random vectors of dimension six. The model is specified in the following way. There are 35 observations, and so t runs from 1 to 35 for the actual observations. However, for this model, it was felt more convenient to initialise the model at $t = 0$. The initial state of the system was specified as $\mathrm{E}(\theta_0) = (50, 50, 50, 50, 50, 50)^{\mathrm{T}}$ and $\mathrm{Var}(\theta_0) = diag(900, 900, 900, 900, 900, 900)$. The covariance

*Note that changes in expectation smaller than expected can still be of concern, since they are indicative of a possible under-utilisation of prior information.

structure for the noise terms was specified to be $\mathrm{Var}(\nu_t) = V,\ \mathrm{Var}(\omega_t) = W$, where

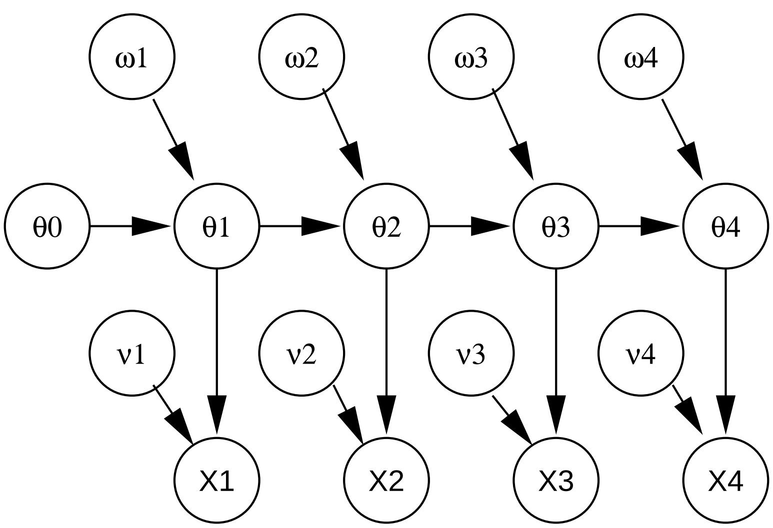
$$V = \begin{pmatrix} 2420.36 & 387.33 & 20.39 & 165.27 & 44.56 & 58.61 \\ 387.33 & 263.85 & 3.85 & 71.51 & 23.48 & 3.27 \\ 20.39 & 3.85 & 30.79 & 3.58 & 1.26 & 5.99 \\ 165.27 & 71.51 & 3.58 & 139.72 & 23.12 & 11.33 \\ 44.56 & 23.48 & 1.26 & 23.12 & 50.01 & 4.78 \\ 58.61 & 3.27 & 5.99 & 11.33 & 4.78 & 44.21 \end{pmatrix}$$

$$W = \begin{pmatrix} 1112.49 & 272.47 & 22.52 & 66.45 & 31.56 & 27.84 \\ 272.47 & 195.50 & 11.53 & 30.07 & 18.51 & 15.37 \\ 22.52 & 11.53 & 29.64 & 5.54 & 4.67 & 6.28 \\ 66.45 & 30.07 & 5.54 & 78.91 & 14.04 & 8.03 \\ 31.56 & 18.51 & 4.67 & 14.04 & 40.50 & 7.32 \\ 27.84 & 15.37 & 6.28 & 8.03 & 7.32 & 32.97 \end{pmatrix}$$

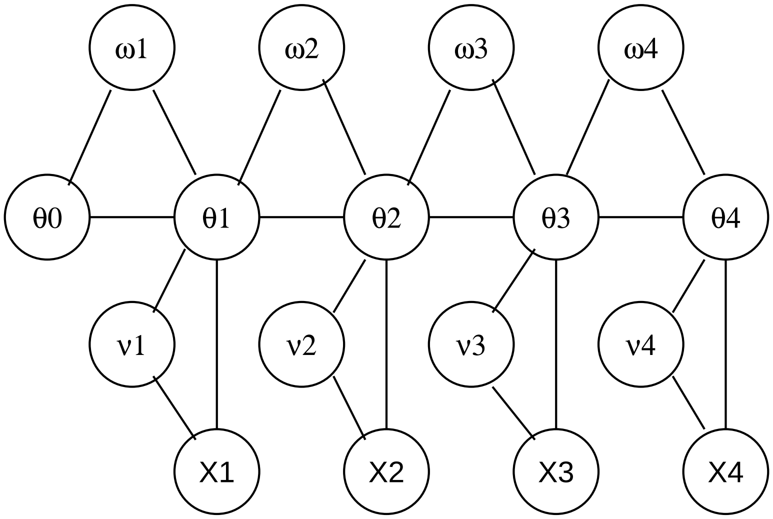
See [9] for an explanation of the given specification. These specifications determine the model, but note that there are $6 \times 4 \times 35 + 6 = 846$ variables in this problem (assuming that we are interested in the noise terms). This problem is about at the limit of the size which can be tackled by a brute force approach, making a local computation approach particularly attractive.

3.2 Graphical models

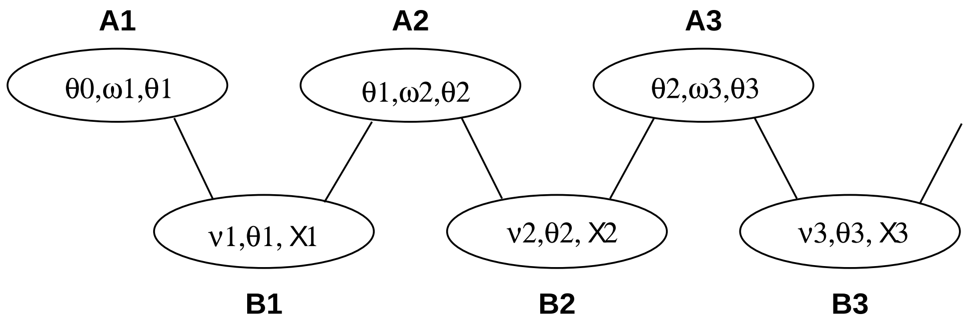
We are interested in making inferences about the noise terms in this example (in order to help diagnose deficiencies of the model), and so the noise terms need to be included in the model. The first part of the DAG for this structure is therefore as follows (note that the DAG nodes are all multivariate).



Marrying parents and dropping arrows gives the moral graph for the problem (note that the moral graph nodes are all multivariate).



Again we are fortunate in the sense that the moral graph is ready-triangulated, and so the clique-tree can be directly constructed as follows.



3.3 Some constants

The code for such a problem can be constructed as follows. First, the data is read, turned into a matrix, and some constants are defined.

```
(def mydata (read-data-columns "ex-mdlm.dat"))
(def data (make-array ’(6 35) :initial-contents mydata))

(def v
#2a(
    (  2420.36       387.330       20.3907       165.274       44.5645       58.6081    )
    (  387.330       263.850       3.85480       71.5054       23.4794       3.26543    )
    (  20.3907       3.85480       30.7910       3.58376       1.25836       5.98943    )
    (  165.274       71.5054       3.58376       139.715       23.1193       11.3268    )
    (  44.5645       23.4794       1.25836       23.1193       50.0087       4.78345    )
    (  58.6081       3.26543       5.98943       11.3268       4.78345       44.2135    )
   )
)

(def w
#2a(
    (  1112.49       272.473       22.5176       66.4472       31.5611       27.8440    )
    (  272.473       195.499       11.5298       30.0701       18.5065       15.3737    )
    (  22.5176       11.5298       29.6411       5.53985       4.66797       6.27591    )
    (  66.4472       30.0701       5.53985       78.9076       14.0421       8.03411    )
    (  31.5611       18.5065       4.66797       14.0421       40.5035       7.32038    )
    (  27.8440       15.3737       6.27591       8.03411       7.32038       32.9728    )
   )
)
```

```
(def e0 (coerce (repeat 50 6) 'array))
(def ee0 (coerce (append (repeat 50 12) (repeat 0 6)) 'array))
(def w0 (diagonal (repeat 900 6)))
(def zero66 (diagonal (repeat 0 6)))
```

These are all self-explanatory.

3.4 Defining the clique-tree

We can now create the type B cliques as follows.

```
(dolist (i (iseq 1 35))
 (create-tree-node
  (intern (format nil "b~a" i))
  (list (intern (format nil "x1.~a" i))
(intern (format nil "x2.~a" i))
(intern (format nil "x3.~a" i))
(intern (format nil "x4.~a" i))
(intern (format nil "x5.~a" i))
(intern (format nil "x6.~a" i))
(intern (format nil "theta1.~a" i))
(intern (format nil "theta2.~a" i))
(intern (format nil "theta3.~a" i))
(intern (format nil "theta4.~a" i))
(intern (format nil "theta5.~a" i))
(intern (format nil "theta6.~a" i))
(intern (format nil "nu1.~a" i))
(intern (format nil "nu2.~a" i))
(intern (format nil "nu3.~a" i))
(intern (format nil "nu4.~a" i))
(intern (format nil "nu5.~a" i))
(intern (format nil "nu6.~a" i))
)
  ee0
  (bind-rows (bind-columns
      (+ w0 (* w i) v)
      (+ w0 (* w i))
      v)
     (bind-columns
      (+ w0 (* w i))
      (+ w0 (* w i))
      zero66)
     (bind-columns
      v
      zero66
      v)
     )
  (if (= i 35)
      (list (intern (format nil "a~a" i)))
    (list (intern (format nil "a~a" i))
  (intern (format nil "a~a" (+ i 1))))
    )
))
```

Note that the expression `(intern (format nil "b~a" i))` means “create the Lisp symbol `b`*i*, where *i* is a variable”. This trick is used a lot for the construction of big models with a repetitive structure. Next, the type A cliques can be constructed, in a very similar way.

```
(dolist (i (iseq 1 35))
 (create-tree-node
  (intern (format nil "a~a" i))
  (list (intern (format nil "theta1.~a" (- i 1)))
        (intern (format nil "theta2.~a" (- i 1)))
        (intern (format nil "theta3.~a" (- i 1)))
        (intern (format nil "theta4.~a" (- i 1)))
        (intern (format nil "theta5.~a" (- i 1)))
        (intern (format nil "theta6.~a" (- i 1)))
        (intern (format nil "theta1.~a" i))
```

```
        (intern (format nil "theta2.~a" i))
        (intern (format nil "theta3.~a" i))
        (intern (format nil "theta4.~a" i))
        (intern (format nil "theta5.~a" i))
        (intern (format nil "theta6.~a" i))
(intern (format nil "omega1.~a" i))
(intern (format nil "omega2.~a" i))
(intern (format nil "omega3.~a" i))
(intern (format nil "omega4.~a" i))
(intern (format nil "omega5.~a" i))
(intern (format nil "omega6.~a" i))
)
  ee0
  (bind-rows (bind-columns
      (+ w0 (* w (- i 1)))
      (+ w0 (* w (- i 1)))
      zero66)
     (bind-columns
      (+ w0 (* w (- i 1)))
      (+ w0 (* w i))
      w)
     (bind-columns
      zero66
      w
      w)
     )
  (if (= i 1)
      (list (intern (format nil "b~a" i)))
    (list (intern (format nil "b~a" i))
  (intern (format nil "b~a" (- i 1))))
    )
))
```

3.5 Defining the moral graph

Next, moral graph nodes need to be created, for diagnostic plotting purposes. Since there isn't room on the average computer screen for the moral graph for all 35 time point, the structure will only be constructed for the first 8 time points only.

```
;; number of moral nodes to create and plot
(def plotnum 8)
(create-moral-node (intern "theta.0")
   (list (intern "theta1.0")
 (intern "theta2.0")
 (intern "theta3.0")
 (intern "theta4.0")
 (intern "theta5.0")
 (intern "theta6.0"))
   (intern "a1")
   "Theta(0)"
   (list (intern "theta.1")
 (intern "omega.1"))
   )
(send (symbol-value (intern "theta.0")) :location (list (/ 1 (+ plotnum 2)) 0.4))
(dolist (i (iseq 1 plotnum))
  ;; create the theta node
  (create-moral-node (intern (format nil "theta.~a" i))
    (list (intern (format nil "theta1.~a" i))
  (intern (format nil "theta2.~a" i))
  (intern (format nil "theta3.~a" i))
  (intern (format nil "theta4.~a" i))
  (intern (format nil "theta5.~a" i))
  (intern (format nil "theta6.~a" i)))
    (intern (format nil "b~a" i))
    (format nil "Theta(~a)" i)
    (if (< i plotnum)
(list (intern (format nil "omega.~a" i))
      (intern (format nil "nu.~a" i))
      (intern (format nil "x.~a" i))
```

```
      (intern (format nil "theta.~a" (- i 1)))
      (intern (format nil "theta.~a" (+ i 1)))
      (intern (format nil "omega.~a" (+ i 1)))
      )
      (list (intern (format nil "omega.~a" i))
    (intern (format nil "nu.~a" i))
    (intern (format nil "x.~a" i))
    (intern (format nil "theta.~a" (- i 1)))
    )
      )
    )
  (send (symbol-value (intern (format nil "theta.~a" i))) :location
(list (* (+ i 1) (/ 1 (+ plotnum 2))) 0.4)
)
  ;; create the omega node
  (create-moral-node (intern (format nil "omega.~a" i))
     (list (intern (format nil "omega1.~a" i))
   (intern (format nil "omega2.~a" i))
   (intern (format nil "omega3.~a" i))
   (intern (format nil "omega4.~a" i))
   (intern (format nil "omega5.~a" i))
   (intern (format nil "omega6.~a" i)))
     (intern (format nil "a~a" i))
     (format nil "Omega(~a)" i)
     (list (intern (format nil "theta.~a" i))
   (intern (format nil "theta.~a" (- i 1))))
     )
  (send (symbol-value (intern (format nil "omega.~a" i))) :location
(list (* (+ i 0.5) (/ 1 (+ plotnum 2))) 0.2)
)
  ;; create the nu node
  (create-moral-node (intern (format nil "nu.~a" i))
     (list (intern (format nil "nu1.~a" i))
   (intern (format nil "nu2.~a" i))
   (intern (format nil "nu3.~a" i))
   (intern (format nil "nu4.~a" i))
   (intern (format nil "nu5.~a" i))
   (intern (format nil "nu6.~a" i)))
     (intern (format nil "b~a" i))
     (format nil "Nu(~a)" i)
     (list (intern (format nil "theta.~a" i))
   (intern (format nil "x.~a" i)))
     )
  (send (symbol-value (intern (format nil "nu.~a" i))) :location
(list (* (+ i 0.5) (/ 1 (+ plotnum 2))) 0.6)
)
  ;; create the x node
  (create-moral-node (intern (format nil "x.~a" i))
     (list (intern (format nil "x1.~a" i))
   (intern (format nil "x2.~a" i))
   (intern (format nil "x3.~a" i))
   (intern (format nil "x4.~a" i))
   (intern (format nil "x5.~a" i))
   (intern (format nil "x6.~a" i)))
     (intern (format nil "b~a" i))
     (format nil "X(~a)" i)
     (list (intern (format nil "theta.~a" i))
   (intern (format nil "nu.~a" i)))
     )
  (send (symbol-value (intern (format nil "x.~a" i))) :location
(list (* (+ i 1) (/ 1 (+ plotnum 2))) 0.8)
)
)
```

The plots can now be created in the usual way.

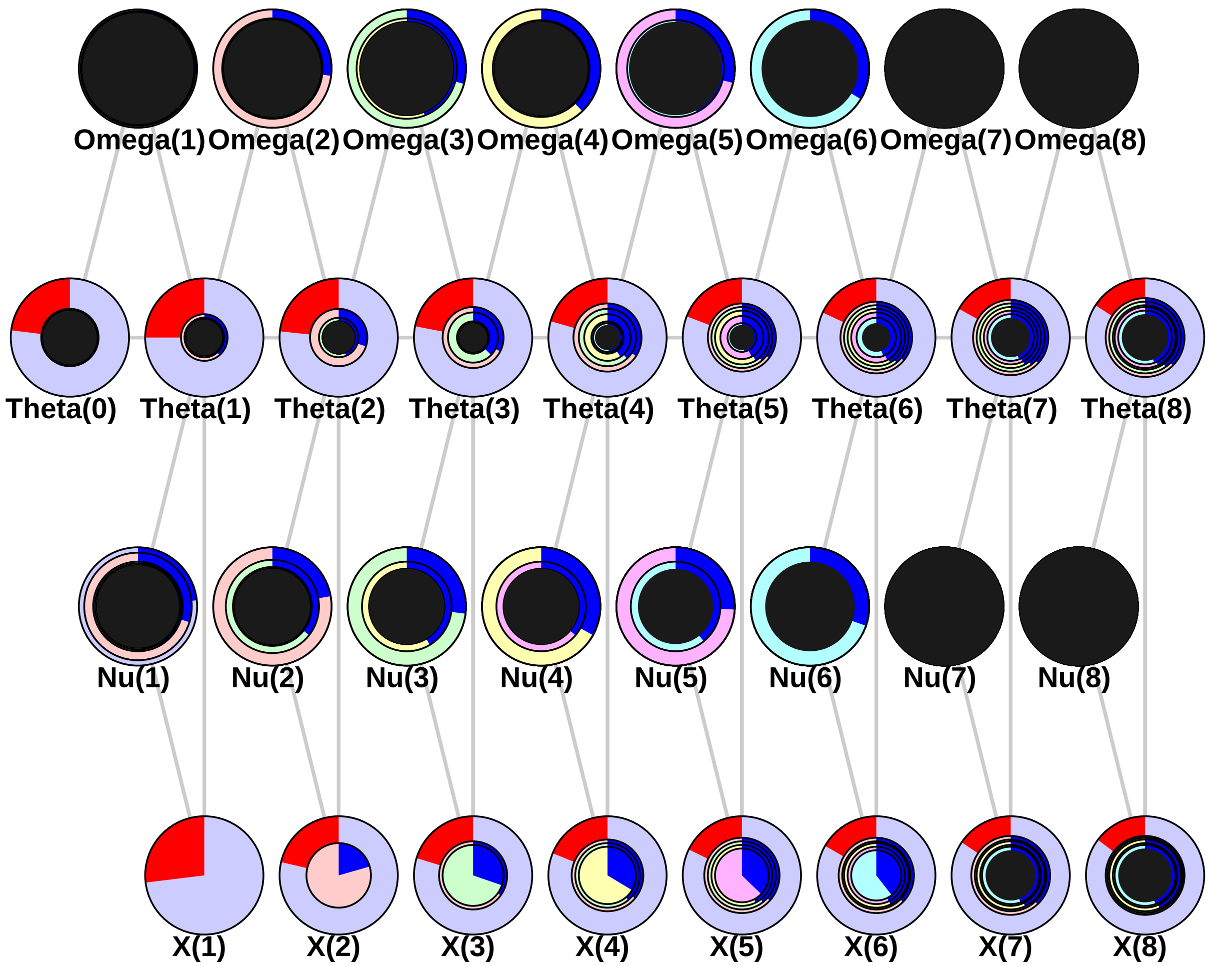
```
(create-moral-plot ’myplot)
(create-global-moral-plot ’myplot2)
```

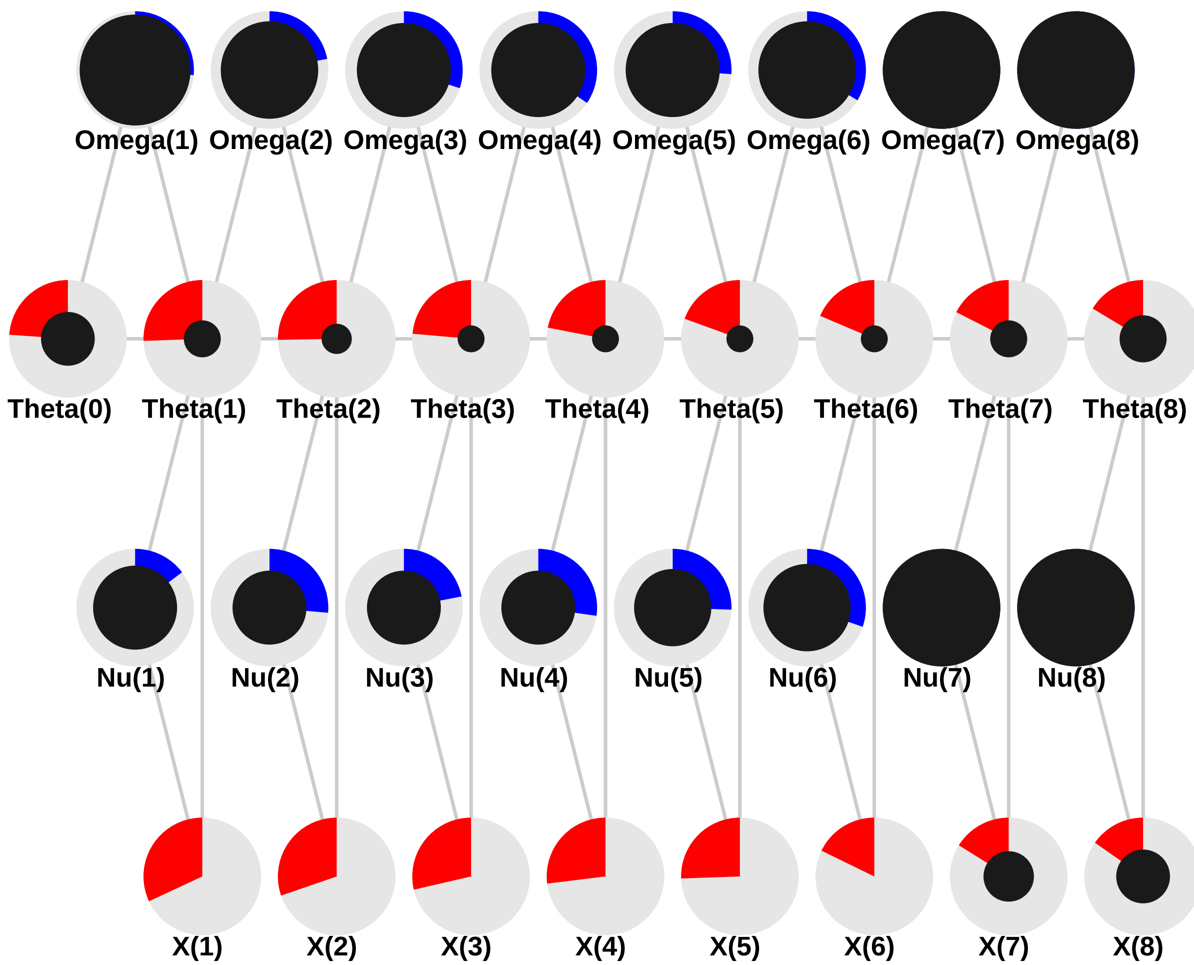
3.6 Adjustments

The first 6 weeks of observations will be added into the model.

```
;; Sequentially introduce the data
(dolist (i (iseq 1 6))
(format t "~&Data for week ~a" i)
(send (symbol-value (intern (format nil "b~a" i))) :observe
      (list
            (intern (format nil "x1.~a" i))
            (intern (format nil "x2.~a" i))
            (intern (format nil "x3.~a" i))
            (intern (format nil "x4.~a" i))
            (intern (format nil "x5.~a" i))
            (intern (format nil "x6.~a" i))
    )
      (select (column-list data) (- i 1))
      )
(send myplot :record)
(send (symbol-value (intern (format nil "b~a" i))) :absorb)
)
```

The resulting plot windows give a good impression of the adjustment process, and the way information flows forward and backwards through time in such models.


Omega(1)
Omega(2)
Omega(3)
Omega(4)
Omega(5)
Omega(6)
Omega(7)
Omega(8)
Theta(0)
Theta(1)
Theta(2)
Theta(3)
Theta(4)
Theta(5)
Theta(6)
Theta(7)
Theta(8)
Nu(1)
Nu(2)
Nu(3)
Nu(4)
Nu(5)
Nu(6)
Nu(7)
Nu(8)
X(1)
X(2)
X(3)
X(4)
X(5)
X(6)
X(7)
X(8)



The file included as part of the distribution then goes on to extract the adjusted expectations of the residuals, and plot them on a line graph. See the example source file for more details.

4 Important note/Disclaimer

It is important to note that BAYES-LIN is a rapidly developing prototype system, and does contain many bugs. You should not rely on BAYES-LIN producing correct output, and should verify parts of calculations as far as possible, using alternative software, such as $[B/D]$ (see [10]). The author accepts no liability whatsoever regarding the use of BAYES-LIN, errors or losses arising from the use of BAYES-LIN, *etc*. Feel free to check the source code, correct it, and email the corrections to the author.

5 Command reference

This section lists all global variables, functions, object prototypes and methods defined by the BAYES-LIN system. On-line help is available. To obtain help on a global function, *eg.* `create-moral-node` use the expression `(help 'create-moral-node)` To obtain help for a method, *eg.* the `:observe` method of the `moral-node-proto` object, use the expression `(send moral-node-proto :help :observe)`.

5.1 Global variables

Variable	Description
`*tree-nodes*`	A list of symbols representing instances of tree-node objects created using the `create-tree-node` global function.
`*moral-nodes*`	A list of symbols representing instances of moral-node objects created using the `create-moral-node` global function.

5.2 Global functions

Function	Description
`create-tree-node`	A function to create and initialise clique-tree objects.
`create-moral-node`	A function to create and initialise moral graph objects.
`create-moral-plot`	A function to create a graphics window for illustrating and diagnosing Bayes linear adjustments.
`ginv`	Function to return the Moore-Penrose generalised inverse of a real square symmetric matrix.

5.3 Object prototypes

Object prototype	Description
`tree-node-proto`	The prototype for objects representing clique-tree nodes.
`moral-node-proto`	The prototype for objects representing moral graph nodes.
`moral-plot-proto`	The prototype for a graphics window object for the displaying of information relating to current Bayes linear adjustments.
`global-moral-plot-proto`	A plot to summarise the partial adjustments shown on the `moral-plot-proto` plots.

5.3.1 `tree-node-proto` slots and methods

Slot	Description
`name`	The name of the object.
`variables`	List of random variables associated with this tree-node object.
`neighbours`	List of neighbouring junction tree nodes.
`variance`	Variance matrix associated with the variable list.
`expectation`	Expectation vector associated with the variable list.
`var-d-inv`	$\mathrm{Var}(D)^{-1}$
`cov-d-self`	$\mathrm{Cov}(D,\cdot)$
`obs-vars`	D
`obs-d-ed`	$d - \mathrm{E}(D)$
`location`	On a $(0,1)\times(0,1)$ scale for plotting.

Method	Description
`:absorb`	Absorb information from last `:observe` ready for next observe.
`:aex`	Adjusted expectation.
`:avar`	Adjusted variance.
`:cov`	Current covariance.
`:ex`	Current expectation.
`:info`	Prints some information relating to the object.
`:location`	Accessor method.
`:observe`	Method to introduce data to the tree.
`:positions`	Variable positions.
`:propagate`	Method used to propagate information around the tree.
`:remove-neighbour`	Remove a neighbour from the list.
`:resolution`	Partial resolution for the current adjustment.
`:rvar`	Resolved variance matrix.
`:size-ratio`	Partial size-ratio for the current adjustment.
`:transform`	Partial resolution transform for the current adjustment.
`:var`	Current variance matrix.

5.3.2 `moral-node-proto` slots and methods

Slot	Description
`name`	Name of the object.
`variables`	List of random variables associated with this moral node object.
`tree-node`	Name of a tree-node which contains all of the variables in this moral node.
`print-name`	A string for plotting purposes.
`neighbours`	A list of neighbouring moral graph nodes.
`location`	On a $(0,1)\times(0,1)$ scale for plotting.
`var_b_inv`	Inverse of the *a priori* variance matrix for the variables represented by this node.
`ex_b`	The prior expectation vector for this node.
`resolutions`	List of resolutions for the partial adjustments.
`size-ratios`	List of partial size-ratios for the adjustments.
`global-size-ratios`	List of global size-ratios for the adjustments.

Method	Description
`:absorb`	Absorb info ready for next `:observe`.
`:aex`	Adjusted expectation vector.
`:avar`	Adjusted variance matrix.
`:bearing`	Bearing vector.
`:ex`	Current expectation vector.
`:info`	Prints some info about the object.
`:location`	Accessor method.
`:observe`	Introduce data into the graph.
`:remove-neighbour`	Remove a node from the neighbour list.
`:resolution`	Partial resolution wrt *a priori* structure.
`:rvar`	Partial resolved variance matrix.
`:size-ratio`	Partial size-ratio.
`:global-size-ratio`	Global size-ratio.
`:transform`	Partial resolution transform.
`:tree-node`	Accessor method.
`:var`	Current variance matrix.

5.3.3 `moral-plot-proto` slots and methods

This object inherits all slots and methods from `graph-window-proto`, but also has the following.

Slot	Description
`nodes`	List of nodes to be plotted.
`real-size`	Window size.
`radius`	Node radius (scaled).
`diagnostics`	Flag for diagnostic plotting.
`node-labels`	Flag for node label printing.
`outlines`	Flag for node outline printing.

Method	Description
`:plot-arcs`	Draw the arcs associated with a given node.
`:plot-node`	Draw the given node, and all its shadings.
`:r-to-s`	Take “real” (screen) coords to scaled coords.
`:record`	Record current adjustment information for inclusion in the plot.
`:redraw`	Guess!
`:resize`	Recalculate scale parameters.
`:s-to-r`	Scaled to real coord transform.
`:sr-map`	Function which maps $(0,\infty)$ to $(-1,1)$ monotonically, mapping 1 to 0. This is the function used to transform size ratios for red and blue diagnostic shadings.
`:diagnostics`	Set and unset diagnostics plotting.
`:node-labels`	Set and unset node label printing.
`:outlines`	Set and unset node label printing.

5.3.4 `global-moral-plot-proto` slots and methods

This object inherits all slots and methods from `moral-plot-proto` and has no others.

References

[1] M. Farrow and M. Goldstein. Bayes linear methods for grouped multivariate repeated measurement studies with application to crossover trials. *Biometrika*, 80(1):39–59, 1993.

[2] M. Goldstein. Revising previsions: a geometric interpretation. *J. R. Statist. Soc.*, B:43:105–130, 1981.

[3] M. Goldstein. Revising exchangeable beliefs: subjectivist foundations for the inductive argument. In P. Freeman and A.F.M. Smith, editors, *Aspects of Uncertainty: A Tribute to D. V. Lindley*. Wiley, 1994.

[4] M. Goldstein. Bayes linear methods I - Adjusting beliefs: concepts and properties. Technical Report 1995/1, Department of Mathematical Sciences, University of Durham, 1995.

[5] M. Goldstein. Prior inferences for posterior judgements. In M. L. D. Chiara et al., editors, *Structures and norms in science*. Pordrecht Kluwer, 1997.

[6] M. Goldstein and D. A. Wooff. Bayes linear computation: concepts, implementation and programming environment. *Statistics and Computing*, 5:327–341, 1995.

[7] J. Pearl. *Probabilistic reasoning in intelligent systems*. Morgan Kaufmann, 1988.

[8] L. Tierney. *LISP-STAT: An object oriented environment for statistical computing and dynamic graphics*. Wiley, 1990.

[9] D. J. Wilkinson and M. Goldstein. Bayes linear covariance matrix adjustment for multivariate dynamic linear models. *U. Ncle. Stats. Tech. report STA97,12*, `http://www.ncl.ac.uk/~nstat/preprints/`, 1997.

[10] D. A. Wooff and M. Goldstein. [B/D] — Beliefs adjusted by Data: Bayes linear methods programming language. *Internet site* `http://fourier.dur.ac.uk:8000/stats/bd/`, 1995.